REUSABILITY IN SOFTWARE ENGINEERING *

Veikko Seppänen †

Technical Report #87-05

January 27, 1987

Department of Information and Computer Science
University of California, Irvine
Irvine, CA92717, U.S.A.

*This research has been supported by grants from The Technical Research Centre of Finland, The Finnish Cultural Foundation, The United States Educational Foundation in Finland/Asla Fulbright Committee, The Foundation of Jenny and Antti Wihuri, The Association of Finnish Electrical Engineers, and by the Advanced Software Engineering Project at the University of California, Irvine.

†On leave from The Technical Research Centre of Finland, Computer Technology Laboratory, P.O. Box 181, SF-90101, Oulu, Finland

Abstract

This paper surveys recent work concerning reusability in software engineering. The current directions in software reusability are discussed, and the two major approaches of *reusable building blocks* and *reusable patterns* studied. An extensive bibliography, parts of which are annotated, is included.

Introduction

The purpose of this short survey is to gather information in the loosely coupled field of *reusability in software engineering*. This looseness is due to the different viewpoints taken as far as software reuse is concerned: many new software engineering paradigms include reusability in some form as one of the important issues, but not necessarily as the major one. Examples of these are object-oriented systems, program transformation systems and rapid prototyping [Horowitz84].

The ultimate purpose of software reusability is to increase productivity, but also to increase the quality of the resulting programs [Freeman83] - there are thus strong economic implications involved. A synonym for the phrase *reusable software* might be *capital goods* as suggested by Wegner [Wegner84], since software production is a capital-intensive process in which both human and technical resources can be reused to achieve the stated economic goals.

Although reusability is not a new idea, there have been no major breakthroughs within the past several years to establish practical, beneficial reusability schemes in the software industry [Freeman83]. This is due to both the difficulties in implementing true production environments for *reusable code* that could successfully support classifications, storage and retrieval of reusable components [Prieto-Diaz85], and in constructing production-quality versions of new software engineering paradigms that support active reusable patterns of the production process rather than passive reusable building blocks [Neighbors80].

The basic concepts of software reusability, classifications of different reuse schemes, and analyses of the proposed paradigms within these schemes will be studied in the following sections. To our knowledge there exists a short survey in the same field [Sundfor83], another that reviews reusable software implementation technologies for specific software development needs [Grabow84], and the general articles included in this tutorial. The goal of this survey is, however, to study some more recently published papers, as well as to provide a concise framework for the concepts of reuse in software engineering in general.

Software reusability

The phrase *reusable software engineering* [Freeman83] emphasizes the reuse of all information generated during the software development process: the questions *what* and *how* to reuse in *which* way that is most successful, arise. This concerns both taking the full advantage of reusable information and intentionally producing information that can be reused in the future.

Classifications and directions of software reusability

Freeman [Freeman83] classifies the reusable forms of information related to the process of software engineering into *code fragments, logical structures, functional architectures, external knowledge* and *environment-level knowledge*. A code fragment can be a piece of a program in a programming language, but also a piece of an executable specification. Logical structures form the internal design of the system, as opposed to external designs of functional architectures in the form of functional collections or generic systems [Freeman83]. The external knowledge is classified by Freeman into *application-area* and *development* knowledge, and the environmental-level information into *utilization* and *technology-transfer* knowledge.

Freeman proposes a set of long-term (longer than 5 years) research directions relevant to effective software reuse. These directions are the following [Freeman83]:

1. *Code fragments:* Development of advanced component technologies including consistent classification schemes, packaging and quality assurance standards and component delivery mechanisms.

2. *Logical structures:* Architecture visualizations extended to new dimensions from simple graphical representations.

3. *Functional architectures:* Development of domain analysis techniques to enhance identification of functional collections, and new system generation technologies that make it possible to generate rapidly similar software systems.

4. *External knowledge:* Formalizations of domains of knowledge and integration of different software engineering paradigms to form effective approaches as far as reusability is concerned.

5. *Environmental knowledge:* Identifications of structural changes in the software production organizations, as well as understanding of programmers' reactions to new paradigms.

According to Wegner [Wegner84] reusability equates to capital-intensive software goods, software production tools and other resources, including software engineers. Wegner claims this insight to provide real means for enhancing productivity and reliability in software technology. As the expense of hardware is decreasing, the productive use of people and software resources is the key of effective reuse. Wegner's approach to reusability is based on reuse within domains of projects, as he thinks there aren't many benefits of reusability between different projects and products. Also, he believes that reusability of application-independent tools contribute more than reuse of domain-dependent software components.

Reducibility of a problem means that the techniques used to solve it can be reused when dealing with another problem; reducibility is thus a synonym for *reusability*. *Equivalency* is two-way reducibility, and one of the main purposes of effective automated reusability should thus be a goal-directed equivalency-preserving navigation through the software production process. The controlled use of both *abstraction* and *specialization* when navigating through these equivalence classes is necessary.

Wegner also states that as the input-output relation realized by a function is an invariant for all programs that realize the function, a uniform reusable rule of computation in the function domain for all arguments is actually defined by an invariant. The notion of *invariancy* is thus another synonym for reusability.

Horowitz and Munson [Horowitz84] have studied different forms of the concept of reusable software and evaluated them from the viewpoint of the development of large software systems. In addition to reusable code in the form of subroutine libraries, compilers, simulations and parameterized systems, the spectrum of reusability can be widened, according to them, by including more general forms of software component libraries, very high level program producing systems that support reusable designs, reusable processors, program transformations, application generators and prototyping. Furthermore, such new programming languages as Ada have concepts that support reusability, and, for instance, spreadsheet programming may be considered as reusing templates.

Jones [Jones84] classifies reusable software into reusable data, reusable architectures, reusable designs, common systems, reusable programs and reusable modules. He claims that reusable code needs major efforts in the other reusability areas, before it can become a sound basic technology. A related view is shared by Ramamoorthy et al. [Ramamoorthy86], who argue that although studies have shown the importance of software reusability, the areas of the role of managers in promoting reusability, reusability of specifications and designs, reusability metrics and the role of AI techniques in enhancing reusability haven't received sufficient attention so far.

The management incentive is emphasized also by Jones et al. in [Jones85] and Clapp [Clapp84], and the role of knowledge-based techniques are emphasized by Barbuceanu [Barbuceanu86], Green et al. [Green83], Arango [Arango85] and Barra et al. [Barra86]. Barbuceanu proposes the use of high-level domain descriptions in terms of structured object descriptions, and the fundamental activities of semantic editing, application-driven simplification and compilation into a conventional programming language to support software reusability. Green et al. define as a long-term goal in knowledge-based software engineering the development of a sophisticated *reusability facet* or agent to coordinate reuse in a knowledge-based software development environment. Arango offers a framework for a knowledge-based perspective of the process of software construction. His approach is based on conceptual analysis and modelling of domain knowledge: domains of interest are organized into networks that support reusability of *domain analysis* and *domain design*. Barra et al., on the other hand, propose an approach in which a set of knowledge-based tools is used in assisting the reuse of code fragments.

Kandt has developed a prototype of a tool called Pegasus that supports the reuse of software designs in a simple database retrieval manner [Kandt84]. Biggerstaff presents a "radical" hypothesis that reusability is *the essence of design* [Biggerstaff84-b]. Doberkat et al. [Doberkat83] have made experiments on the reusability of design for complex programs by using an optimizer for the SETL language.

Such different reuse schemes as direct reuse without modifications and reuse of abstract modules after refinements, have different implications in software technology, as pointed out in [Standish84]. [Lubars86] defines the in-house type code reuse without major modifications to belong in the class of *reuse in the small*, as opposed to *reuse in the large* that tries to share the reusable artifacts within a large community of applications and users.

The problem areas of practical reuse paradigms include information retrieval, software generators, component composition approaches, program understanding and reuse benefit analyses [Standish84]. Standish emphasizes the importance of realistic expectations as far as software reusability is concerned: he thinks that it is not likely that useful arts of reuse would emerge in any arbitrary and uncultivated software engineering subfields.

Soloway and Ehrlich [Soloway84] have investigated the knowledge types used by experienced programmers. They claim that expert programmers use two basic mental models to enhance reuse, *programming plans* that are generic program fragments of stereotyping action sequences and *rules of programming discourse* that govern the composition of plans into programs.

In the following we are going to use the basic classification of *reusable patterns* and *reusable building blocks* when studying more closely the software reusability paradigms being recently researched and developed [Biggerstaff84]. As suggested by Baltz et al. [Baltz83], the building blocks' approach might be seen as the short-term goal for effective software reuse, and the reusable patterns' approach as the long-term goal that relies on such new software development paradigms as transformation systems, very high-level languages (VHLLs) and knowledge-based techniques.

Reusable patterns

A reusable pattern is more an active element used to generate the target software system than a passive building block used when constructing it from the com-

ponents - it is a matter of manipulation rather than execution to reuse these patterns [Biggerstaff84]. The patterns may be code inside an application generator-like system or inside transformation rules in a transformation system. They are reused when reactivated to drive the generation or transformation process.

Biggerstaff and Perlis [Biggerstaff84] argue that there are three basic classes of systems based on reusable patterns: *language-based systems* including both very high-level languages and problem-oriented languages (POLs), *application generators*, and *transformation systems.*

Cheng et al. [Cheng84] have studied experimentally the use of the VHLL Model [Prywes79], and a program generator as tools for developing management system software without any involvement of professional programmers. The statistics from their experiment show an increase in productivity over the development of the same program manually by professional programmers.

The reusable patterns in application generators are typically encoded knowledge about specific domains of interest that can be reused when producing similar systems in that domain. The Draco system, to be discussed later, is classified into application generators by Biggerstaff and Perlis [Biggerstaff84], although it uses also high level domain-dependent languages and a set of transformations, and has elsewhere been classified into transformation systems [Partsch83]. A recent survey of application generators is made by Horowitz et al. [Horowitz85]: the authors propose a programming language to be extended to include application generator features in order to correct the fact that current application generators lack computational flexibility.

Transformation systems are based on the idea of describing first the target system in an easy to understand, easy-to-use language and then refining it into an executable, efficient target program. As classified by Partsch and Steinbrüggen [Partsch83], there is a general framework of the subclasses of *general transformation systems, special purpose systems* and *general programming environments.* Reusable patterns in transformation systems are most often embedded into the transformation rules, i.e. it is possible to produce similar systems by reusing the proper set of transformations. Arango et al. have used a transformational reusability support system to port the system itself into another target environment - they claim that the approach is a very powerful transformation based maintenance model that allows an undocumented source program to be ported without any modifications into another environment, where it can be reused [Arango86].

Boyle and Muralidharan [Boyle84] present a system transforming pure Lisp programs into Fortran code, where the Lisp program is seen as an abstract specification for the Fortran version. Transformation rules include many reusable patterns for Lisp to Fortran translations, but no more broadly reusable information for the software development process. Cheatham, on the other hand, suggests transformation systems for a software engineering paradigm [Cheatham84]: An environment supporting the methodology that facilitates the reuse of abstract programs written in a domain-dependent language, which is extended from a base language, has been developed by his group. The abstract programs are transformed into their concrete counterparts by using transformation rules.

The reuse paradigm in Draco

Draco is a paradigm and a tool to facilitate refinements of domain-dependent software specifications into other domain-dependent specifications, and reuse of both *domain analysis results, domain design results* and refined *domain-language code* produced by Draco [Neighbors80], [Neighbors84], [Freeman86].

The main goal in Draco is to support the construction of similar software systems from components by refining one domain representation to another, ultimately to the target programming language. Source-to-source translations within domains are used to optimize transformations.

The reuse of domain analysis is done in Draco when a system is specified in the domain language developed for the specific domain, and the reuse of domain design is done when the domain-dependent components are used to refine the specification into another domain. The concept of *components* in Draco defines the semantics of the domain language in terms of other domains, i.e. they are the reusable *patterns.* A domain description includes definitions for *a parser, a prettyprinter, transformations and components.*

Draco is based on the separation of domains of interest, i.e. domain languages are different for different domains. The idea is to select implementation-independent universes of discourses that cover numbers of similar software systems. Neighbors gives as one example of a domain augmented transition networks [Neighbors80].

The domain-dependency may, however, complicate the task of reusing anything from other domains, since a completely new set of objects, actions and relationships that form the domain language's constructs must then be learned [Horowitz84]. The approach taken by some related systems is to use *wide-spectrum languages* that cover the whole spectrum of software development from specifications to executable code, as opposed to Draco domains where the network of domains (languages) must be used to transform high-level specifications into efficient target language programs.

Reusable building blocks

The components to be reused in the reusable building blocks' approach are atomic, passive fragments of software such as code skeletons, subroutines, functions, programs and Smalltalk-type objects.

Dennis et al. [Dennis86], for example, define the characteristics to be met by reusable (Ada) components:

1. The interface is syntactically and semantically clear,
2. The interface is written at an appropriate level,
3. The component does not interfere with its environment,
4. The component is object-oriented,
5. Actions based on the results of a function type component are made at the next level up,
6. The component incorporates scaffolding,
7. The component exhibits high cohesion and low coupling,
8. The component and its interface are readable by persons another than the author,
9. The component is in balance between specificity and generality,
10. The component is documented sufficiently to be found,
11. The component can be used without change or with minor modifications,
12. The component is insulated from its host/target dependencies and assumptions, and
13. The component is standardized in the areas of invoking, controlling, terminating, error-handling, communication and structure.

The major issue in the reuse of building blocks may consider application component libraries or the composition principles used to construct programs from their building blocks - the two approaches will be studied in the next sections.

Application component libraries

Lanergan and Grasso claim that significant productivity gains can be achieved in business software systems production by reusing directly code fragments in libraries without much emphasis on component specification, archiving and retrieving formalisms [Lanergan84]. Matsumoto, on the other hand, views the software design process at *requirements, design* and *program* levels of abstraction in order to provide a substantial degree of component reusability [Matsumoto84]. He proposes a requirements definition to be made for each existing module at its highest level of abstraction, and a traceability scheme between these presentations and reusable program modules to be established to enhance *understanding* and *leverage* of the components.

Polster [Polster86] defines the concept of a "*B*-program" to describe code fragments used in building *partial systems* that contain only subsets of some more general existing programs. *B*-programs include, besides the code, the information specifying the set of partial systems for which they are relevant. Polster has developed a formal model to describe the construction of partial systems by using *B*-programs.

Ledbetter and Cox propose the message/object programming paradigm as the basis of software component reuse [Ledbetter85], [Cox84]. They use the phrase *software-IC* to describe a reusable software package, a standard *binary* file that is a compilation of a C source program generated by Objective-C, which is claimed to combine aspects of subroutine libraries and Unix filter programs in a more efficient way. Dynamic binding is a prerequisite of using software-IC components.

Component organizing and construction techniques

As noted by Prieto-Diaz and Freeman [Prieto-Diaz86], one of the main difficulties in reusing code fragments is the problem of how to locate and retrieve them effectively. Prieto-Diaz proposes the use of the classification theory as the basis of a classification scheme for code fragments to permit their easy location and reuse [Prieto-Diaz85].

The Unix pipe mechanism can be taken as a mean of organizing collections of software pieces into larger programs, for instance into a structure of software filters [Kernighan84]. Litvintchouk and Matsumoto [Litvintchouk84] suggest a methodology based on formal algebra for achieving the design and management of organizations of reusable components in Ada systems.

Multiple inheritance in object-based systems forms, according to [Wegner84], an important mechanism for system evolution analogous to inherited capital resources in any industrial technology. This inheritance is usable in the organization of object libraries that include knowledge about application domains in constructing compositions from the library objects. Curry and Ayers [Curry84] introduce the concept of *a trait* in an object-based system that provides a way to customize nominally similar objects by multiple-component subclassing for slightly different applications. [Goguen84] proposes new mechanisms in parameterized programming, *theories, views* and *module expressions* that can be used to organize module interface specifications into inheritance structures. The *module interconnection languages* have traditionally tried to address the component interconnection issues by having a formal syntax to describe the interfaces, interconnection operations and resource flow [Prieto-Diaz86-b].

[Wegner86] classifies component reuse accordingly into reusability of interfaces, functions that are abstract operations, data abstractions that are abstract variables and processes that are abstract "computers" operating either concurrently or in a distributed manner. These classes consider different capital goods as the primary reusable resources. According to Wegner, long-lived distributed embedded systems should provide (reusable) component evolution while being used and executed, not only during the development time - this could be done by dynamic linking and creation of the abstract

processes.

If data abstractions are used, they must be protected from concurrent access by process abstractions - this is not provided for instance in Ada. Wegner claims that process abstraction is a more powerful paradigm for effective reuse in large evolving programs than data abstractions. Harandi and Young [Harandi85] describe an approach based on a library of reusable abstract design templates, with a mechanism to store, retrieve and manipulate the templates. A template is an abstract and generic problem solution applicable to a large number of specific cases. Manipulation of templates consists of successive refinements and elaborations of templates corresponding collectively to the solution of the given abstract problem statement. Aoyma et al. [Aoyma86] describe a related approach in the area of telecommunication software, where the templates for service software components are called *paradigms.*

Berzins et al. [Berzins86] propose the use of abstractions as a comprehensive software engineering paradigm, yet embedded into the conventional waterfall-like software development model.

The use of generic software modules, called *generic capabilities*, in software development tool building is suggested by Kuo et al. in [Kuo84]. Their primitive classification of generic capabilities include *generic information structures, generic functions* and *generic knowledge.* The reusability of software tools, in the case of form-oriented software, is studied also by Wartik and Penedo in [Wartik86]. Genericity, as in Ada, and inheritance, as in object-oriented languages, are seen as two alternatives to facilitate reusability by Meyers in [Meyers86]. The statically typed language Eiffel that he has developed includes both a limited amount of genericity and multiple inheritance. The limitations are claimed to be necessary in order to avoid overly redundant and complex concepts in a language that would support full features of both inheritance and genericity.

Other reusability paradigms

Adaptable software is sometimes taken as a synonym for reusable software. However, we would like to differentiate between adaptable and reusable software on the basis of the following list of required software adaptations of real-time, embedded systems [Alexandridis86]:

1. *Retargetable* software for new hardware configurations.
2. *Dynamically adaptable* reconfiguration software.
3. *Portable* environments.
4. *Reusable* software *components.*
5. *Reusable component abstractions.*
6. *Reusable concepts.*

The first three forms of adaptability are not covered by the reusability paradigms presented above. Goguen [Goguen86], Schwederski and Siegel [Schwederski86], and Kartashev and Kartashev [Kartashev86] have discussed adaptability of mission-critical software systems. Roussapoulos and Yeh [Roussapoulos84] describe an adaptable "outside-in" methodology for database design, consisting of four phases that facilitate the adoption of different models and representations as needed.

The idea of *evolutionary software development* instead of the successive development and maintenance phases is implicitly the key feature in such new software engineering paradigms as prototyping, operational specifications and transformational approach [Agresti86]. The role of software reusability in the evolutionary approach has been discussed for instance by Barbuceanu [Barbuceanu86] and Balzer [Balzer84]. Reuse of the evolutionary approach, as pointed out by Balzer, is actually a replay of the software development process, i.e. is based on reusable patterns in the development process.

We do not intend to include these new software development paradigms in our survey - except transformation systems mentioned above - as an extensive collection of related papers and an annotated bibliography is presented recently by Agresti [Agresti86]. Furthermore, a good collection of papers describing different aspects of the knowledge-based software engineering paradigm can be found in [Rich86].

Summary

Reusability in software engineering is a many-sided concept that includes more than the purely technical aspects of how to implement reusability schemes. Basically the different approaches that have been proposed differ from each other as far as the reused software engineering information is concerned.

The classification into reusable patterns and reusable building blocks is interesting, as it represents at the same time also the basic difference between the traditional waterfall-like software development paradigm and the newer paradigms: when understanding of the resulting product is captured in early artifacts that do not have such a fine granularity as the implementation related artifacts, the real efforts can be directed to new forms of synthesis, and not to extensive analysis. The transformational approach is perhaps the best example of this difference, and is strikingly present both in the new reusability paradigms and in the new general software development paradigms.

Although many people think that the new paradigms are the way we should be going, there is still quite a gap between reusability ideas and their efficient and effective implementations.

References

General concepts and surveys

[Agresti86] Agresti, W.W., (ed.), *New Paradigms for*

Software Development. IEEE Computer Society Press, Washington D.C., 1986.

The conventional software life-cycle model based on a waterfall like approach has been criticized and new *paradigms* for software development has been suggested. The tutorial includes an introduction into the evolution and assumptions embedded in the conventional life-cycle model and defines the three new software development paradigms of *prototyping, operational specification* and *transformational implementation.* Papers criticizing the waterfall model are included, as well as papers describing more closely each of the three new paradigms in turn. The tutorial contains an extensive bibliography, annotated by the author.

[Alexandridis86] Alexandridis, N.A., "Adaptable Software and Hardware: Problems and Solutions", *IEEE Computer*, vol. 19, no. 2, Feb. 1986, pp. 29-39.

[Batz83] Batz, J.C., Cohen, P.M., Redwine, S.T. Jr., and Rice, J.R., "The Application-Specific Task Area", [1] *IEEE Computer*, vol. 16, no., 11, Nov. 1983, pp. 78-85.

[Biggerstaff84] Biggerstaff, T., Perlis, A.J., "Foreword",[2] *IEEE Transactions on Software Engineering*, vol. SE-10, no. 5, May 1984, pp. 474-476.

A short summary of the state of the technology in software reusability is given. The framework of *reusable building blocks* and *reusable patterns* is created. Reusable building blocks are passive components to be used in the construction of software, either in the form of simple application component libraries or more generally according to some well-formed principles in organizing and composing them. Reusable patterns are active elements inside the mechanisms supporting construction of software systems rather than passive components to be put together. There are three classes of reusable patterns: language-based systems, application generators and transformation systems.

[Biggerstaff84-b] Biggerstaff, T., "A Radical Hypothesis: Reusability Is the Essence of Design", *Proceedings, Compsac84 Conference*, Silver Spring, Maryland: IEEE Computer Society Press, 1984, p. 474.

[Chandersekaran83] Chandersekaran, C.S., Perriens, M.P., "Towards an assessment of software reusability", *Proceedings, ITT Workshop on Reusability in Programming*, Stratford, Connecticut: ITT Programming, 1983, p. 179.

[Clapp84] Clapp, J., "Software Reusability: A Management View", *Proceedings, Compsac84 Conference*, Silver Spring, Maryland: IEEE Computer Society Press, 1984, p. 479.

[Curtis83] Curtis, B., "Cognitive issues in reusability", *Proceedings, ITT Workshop on Reusability in Programming*, Stratford, Connecticut: ITT Programming, 1983, p. 192.

[Freeman83] Freeman, P., "Reusable Software Engineering: Concepts and Research Directions," *Proceedings, ITT Workshop on Reusability in Programming*, Stratford, Connecticut: ITT Programming, 1983, p. 2.

The phrase *reusable software engineering* is defined to emphasize the reuse of all information generated during the software development process. The reusable forms of information are classified into *code fragments, logical structures, functional architectures, external knowledge* and *environment-level knowledge.* The external-level software engineering knowledge consists of *application-area* and *development* knowledge, and the environmental-level information of *utilization* and *technology-transfer* knowledge. Both short-term, mid-term and long-term research directions relevant to effective software reuse in these classes are proposed. The long-term directions are: development of advanced component technologies for code fragments, architecture visualizations for logical structures, development of domain analysis techniques and new system generation technologies for functional architectures, formalizations of domains of external-level knowledge with integration of different software engineering paradigms, and identifications of structural changes in the software production organizations, as well as understanding of programmers' reactions to new reusability related software paradigms.

[Gerhard83] Gerhard, S., "Reusability lessons from verification technology," *Proceedings, ITT Workshop on Reusability in Programming*, Stratford, Connecticut: ITT Programming, 1983, p. 110.

[Grabow84] Grabow, P.C., Noble, W.B., and Huang, C.-C., (eds.), *Reusable software implementation technology reviews*, Hughes Aircraft Company, Fullerton, California, 1984.

[Horowitz84] Horowitz, E., Munson, J.B., "An Expansive View of Reusable Software," *IEEE Transactions on Software Engineering*, vol. SE-10, no. 5, May 1984, pp. 477-487.

The article studies different forms of the concept *reusable software* and evaluates them from the viewpoint of the development of large software systems. In addition to reusable code in the form of subroutine libraries, compilers, simulations and parameterized systems, the spectrum of reusability can be widened by including more general forms of software component libraries, very high level program producing systems that support reusable designs, reusable processors, program transformations, application generating and prototyping. Furthermore, such new programming languages as Ada have concepts that support reusability, and for instance spreadsheet programming may be considered as reusing templates.

[1] of the DoD's STARS program

[2] of the special issue on reusable software

[Horowitz85] Horowitz, E., Kemper, A., and Narasimhan, B., "A Survey of Application Generators," *IEEE Software*, vol. 2, no. 1, Jan. 1985, pp. 40-54.

The results of a survey of the current application generators is presented. The commercially availably application generators (AG) Nomad, Ramis, Focus, ADF and DbaseII are investigated. The basic components of an AG are described and a generic AG having all the basic components and a Nomad-like syntax is described. The possibility of combining the high-level features of an AG with a general-purpose programming language is analyzed. An attempt to extend a programming language with AG features to correct the fact that AGs lacks computational flexibility is outlined.

[Jones84] Jones, T.C., "Reusability in Programming: A Survey of the State of the Art," *IEEE Transactions on Software Engineering*, vol. SE-10, no. 5, May 1984, pp. 488-493.

The 1984 state of the art in reusable data, reusable architectures, reusable designs, common systems, reusable programs and reusable modules is surveyed. Reusable code needs major efforts in the other reusability areas, before it can become a sound basic technology. An estimate is done that in 1990 half of all code is going to be reused, and the other half will be unique among leading software enterprises.

[Jones84-b] Jones, G., "Software Reusability: Approaches and Issues," *Proceedings, Compsac84 Conference*, Silver Spring, Maryland: IEEE Computer Society Press, p. 476.

[Jones85] Jones, B, Krasner, H., Litvintchouk, S., Mellby, J., Mungle, J., and Willman, H., "Issues in software reusability," ACM SIGAda *Ada Letters*, vol. IV, no. 5, Apr. 1985, pp. 97-99.

[Meyers86] Meyers, B., "Genericity versus Inheritance," *Proceedings, OOPSLA'86 Conference*, ACM SIGPLAN *Notices*, vol. 21, no. 11, Nov. 1986, pp. 391-405.

Genericity, as in Ada, and inheritance, as in object-oriented languages, are alternative techniques for ensuring better extendability, reusability and compatibility of software components. The article studies the two approaches and assesses to what extend each may be simulated in a language supporting only the other. The features in the statically typed language Eiffel that provides a limited form of genericity and multiple inheritance, are presented. Having full inheritance and genericity in the same programming language is found to result in a redundant and overly complex design, and a borderline at unconstrained genericity has thus been put into the Eiffel language.

[Nourani85] Nourani, C.F., Jones, G.A., "Software Reusability – A Perspective," *Proceedings, 18th Annual Hawaii International Conference on Systems Sciences*, Hawaii: 1985, p. 447.

[Partsch83] Partsch, H., Steinbrüggen, R., "Program transformation systems," *ACM Computing Surveys*, vol. 15, no. 3., March 1983, pp. 199-236.

A review and classification of a broad class of program transformation systems is done. A general framework for the classification is discussed, and the systems belonging to some of the subclasses *general transformation systems, special purpose systems* and *general programming environments* are briefly presented.

[Prieto-Diaz86-b] Prieto-Diaz, R., Neighbors, J., "Module Interconnection Languages," University of California, Irvine, Department of Information and Computer Science, RTP058, 1986, 77 p.

[Rich86] Rich, C., Waters, R.C., (eds.), *Readings In Artificial Intelligence and Software Engineering*, Morgan Kauffmann Publishers, Los Altos, California, 1986, 602 p.

[Standish84] Standish, T.A., "An Essay on Software Reuse," *IEEE Transactions on Software Engineering*, vol. SE-10, no. 5, May 1984, pp. 494-497.

Software reuse is explored. Such different schemes as direct reuse without modifications and reuse of abstract modules after refinements, have different implications in software technology. The problem areas of practical reuse paradigms, including information retrieval, software generators, component composition approaches, program understanding and reuse benefit analyses are sketched. The importancy of realistic expectations as far as software reusability is concerned, is emphasized: it is not likely that useful arts of reuse would emerge in arbitrary subfields and in such uncultivated systems as databases in knowledge-based systems.

[Sundfor83] Sundfor, S., "Reusable Software Engineering: A Survey of Concepts and Approaches," University of California, Irvine, Department of Information and Computer Science, RTP017, 1983, 24 p.

The results of a survey of reusability in software engineering are presented. Both reusable code, reusable program abstractions, reusable design and reusable analysis are studied. Varietes of reusability approaches is seen to exist, but not actually effectively used in the software industry. The reusable code is found to be the area closest to practical applications.

[Wegner83] Wegner, P., "Varietes of reusability," *Proceedings, ITT Workshop on Reusability in Programming*, Stratford, Connecticut: ITT Programming, 1983, p. 30.

[Wegner84] Wegner, P., "Capital-intensive software technology," *IEEE Software*, vol. 1, no. 6, July 1984. pp, 7-54.

Different capital-intensive software activities are explored, including software components and programming in the large. Software components are seen as capital-intensive

building blocks of which large programs are constructed. Relations among subprograms, data and process abstractions and object-oriented programming are examined. The role of component libraries is studied, and a taxonomy developed to show the maturation of the component technology. Examples of the reusability of concepts in both theoretical and experimental computer science are given.

[Yeh84] Yeh, R.T., Roussopoulos, N., "Management of reusable software," *Proceedings, IEEE Compcon84*, Silver Spring, Maryland: IEEE Computer Society Press, 1984, p. 311-320.

Techniques

[Aoyma86] Aoyma, M., Suzuki, T., Suzuki, M., and Fujimoto, H., "Development of telecommunication software based on paradigms," *Proceedings, Sixth International Conference on Software Engineering for Telecommunication Switching Systems*, Washington D.C.: IEEE Computer Society Press, 1986, p. 112.

[Barra86] Barra, S., Ghisio, O., Gouthier, P., and Truzzi, S., "An environment providing assistance in code reusability", *Proceedings, Sixth International Conference on Software Engineering for Telecommunication Switching Systems*, Washington D.C.: IEEE Computer Society Press, 1986, p. 221.

[Boyle84] Boyle, J.M., Muralidharan, M.N., "Program Reusability through Program Transformation", *IEEE Transactions on Software Engineering*, vol. SE-10, no. 5, May 1984, pp. 574-588.

A practical approach in transforming pure Lisp programs into Fortran code is introduced. The Lisp program is seen as an abstract specification for the Fortran version. Strategic insights of the transformation process are discussed. Transformations are automated refinements of the source program into target language, according to the strategic decisions. An example of transforming a Lisp program into an efficient Fortran program is given.

[Cavaliere83] Cavaliere, M.J., Archambeault, P.J. Jr., "Reusable code at the Hartford Insurance Group", *Proceedings, ITT Workshop on Reusability in Programming*, Stratford, Connecticut: ITT Programming, 1983, p. 273.

[Cheng84] Cheng, T.T., Lock, E.D., and Prywes, N.S., "Use of Very High Level Languages and Program Generation by Management Professionals", *IEEE Transactions on Software Engineering*, vol. SE-10, no. 5, May 1984, pp. 552-563.

The paper studies the experimental use of a very high-level language (VHLL) and a program generator as tools for developing a management system software without any involvement of professional programmers. Statistics from the experiment are seen to show an increase in productivity over the development of the same program manually by professional programmers. This is claimed to be mainly due to better means of detecting errors before the program was generated, and thus saving in debugging efforts.

[Cox84] Cox, B.J., "Message/Object Programming: An Evolutionary Change in Programming Technology". *IEEE Software*, vol. 1, no. 1, Jan. 1984, pp. 50-61.

[Curry84] Curry, G.A., Ayers, R.M., "Experience with Traits in the Xerox Star Workstation", *IEEE Transactions on Software Engineering*, vol. SE-10, no. 5, May 1984, pp. 519-527.

A trait is a modifier attached to existing object definitions in the Xerox Star (8010) workstation's software that has been written in an object-oriented style. The concept of trait provides a way to customize nominally similar objects for slightly different applications. The article describes the use of traits, and states as experiences that multiple-component subclassing is a useful way to increase sharing in software design and implementation components. The approach needs, however, both a strong language and architectural support in order to be beneficial in economical terms.

[Deutsch83] Deutsch, L.P., "Reusability in the Smalltalk-80 Programming System", *Proceedings, ITT Workshop on Reusability in Programming*, Stratford, Connecticut: ITT Programming, 1983, p. 72.

[Gladney83] Gladney, H.M., "A concise experiment in program reusability", *Proceedings, ITT Workshop on Reusability in Programming*, Stratford, Connecticut: ITT Programming, 1983, p. 207.

[Harandi85] Harandi, M.T., Young, F.H., "Template Based Specification and Design," *Proceedings of the Third International Workshop on Software Specification and Design*, Washington D.C.: IEEE Computer Society Press, 1985, p. 94.

Design aspects of a template based program development environment are described. A template is a reusable, abstract and generic problem solution applicable to a large number of specific situations. Successive refinement and elaboration of templates, guided by the information embedded in them, is used to find the solution for the given abstract statement of the problem. A mechanism to create, store, retrieve and manipulate templates is introduced.

[Kandt84] Kandt, K., "Pegasus: A tool for the acquisition and reuse of software designs,", *Proceedings, Compsac84 Conference*, Silver Spring, Maryland: IEEE Computer Society Press, 1984, p. 288.

[Kernighan84] Kernighan, B.W., "The Unix System and Software Reusability," *IEEE Transactions on Software Engineering*, vol. SE-10, no. 5, May 1984, pp. 513-518.

The Unix operating system is seen to contain several facilities enhancing reuse of software, for instance the pipe

mechanism that supports reuse by the self nature of its operation, but also through its effect on programming conventions. Furthermore, the on line C source code of Unix systems allows the existing programs to be used as models for new ones. The article studies reusability in Unix at the levels of a library, a programming language, a program, a system and a concept.

[Kuo84] Kuo, H.C., Chow, A.L., and Kishimoto, Z., "An Approach to Flexible Software System Design Using Generic Capabilities," *Proceedings, Compsac84 Conference*, Silver Spring, Maryland: IEEE Computer Society Press, 1984, p. 294.

The approach of building flexible, evolving software tools is discussed. In the proposed approach the tools are based on *generic capabilities* which are small flexible tool fragments, to be extended and customized into complete tools. A primitive classification of generic capabilities is given, including *generic information structures, generic functions* and *generic knowledge*. The use of this approach in building software tools is analyzed.

[Lanergan84] Lanergan, R.T., Grasso, C.A., "Software Engineering With Reusable Designs and Code," *IEEE Transactions on Software Engineering*, vol. SE-10, no. 5, May 1984, pp. 498-501.

The article studies reusability in business software systems. Sixty percent of all business application designs and code are seen as redundant software that could be standardized into reusable functional modules and logical structures. The approach based on reusability can have significant productivity gains in the development, and eliminate 60-80 percent of the maintenance problem.

[Ledbetter83] Ledbetter, L., "Reusability of domain knowledge in the automatic programming system Ø", *Proceedings, ITT Workshop on Reusability in Programming*, Stratford, Connecticut: ITT Programming, 1983, p. 97.

[Ledbetter85] Ledbetter, L., Cox, B., "Software-ICs: A plan for building reusable software components," *Byte*, June 1985, pp. 307-316.

[Matsumoto84] Matsumoto, Y., "Some Experiences in Promoting Reusable Software: Presentation in Higher Abstract Levels," *IEEE Transactions on Software Engineering*, vol. SE-10, no. 5, May 1984, pp. 502-513.

The software design process is viewed at several levels of abstraction in order to provide substantial degree of reusability. The levels are requirements, design and program. The article proposes a presentation in a requirements definition style to be made for each existing module at the highest level of abstraction, a traceability scheme between these presentations and reusable program modules to be established. Module descriptions at all abstraction levels are suggested to be done in an Ada-like language. The approach is seen to increase the scope of the concept reusable code.

[McCain85] McCain, R., "A software development methodology for reusable components," *Proceedings, 18th Annual Hawaii International Conference on Systems Sciences*, Hawaii: 1985, p. 319.

[Oskarsson83] Oskarsson, Ö., "Software reusability in a system based on data and device abstractions - A case study", *Proceedings, ITT Workshop on Reusability in Programming*, Stratford, Connecticut: ITT Programming, 1983, p. 160.

[Parnas83] Parnas, D., Clements, P.C., and Weiss, D.M., "Enhancing Reusability with Information Hiding", *Proceedings, ITT Workshop on Reusability in Programming*, Stratford, Connecticut: ITT Programming, 1983, p. 240.

[Prywes79] Prywes, N.S., Pneuli, A., and Shastry, S., "Use of a Nonprocedural Specification Language and Associated Program Generator in Software Development," *ACM Transactions on Programming Languages and Systems*, vol. 1, no. 2, Feb. 1979, pp. 196-217.

Use of the Model II language and the associated program generator is described to illustrate the benefits of a nonprocedural, very high level language in specifying information processing systems. The use of the Model II language makes it easier and shorter to specify business software systems, as procedural and control facilities do not need to be specified. The associated Model II processor is capable of generating PL/I programs from Model II specifications.

[Rice83] Rice, J.R., "Interface issues in a software parts technology", *Proceedings, ITT Workshop on Reusability in Programming*, Stratford, Connecticut: ITT Programming, 1983, p. 129.

[Wartik86] Wartik, S.P., Penedo, M.H., "Fillin: A Reusable Tool for Form-Oriented Software," *IEEE Software*, vol. 3, no. 2, March 1986, pp. 61-69.

Research

[Arango85] Arango, G., "A knowledge engineering perspective on software construction", Department of Information and Computer Science, University of California, Irvine, 1985, 38 p.

The contributions of AI techniques in software construction are discussed. A paradigm of software development based on conceptual analysis and modeling of domain knowledge is presented. Domain theories and languages can be organized into networks that support reusability of domain analysis and design, as well as the integration of program construction and verification knowledge.

[Arango86] Arango, G., Baxter, I., Freeman, P., and Pidgeon, C., "TMM: Software Maintenance by Transformation", *IEEE Software*, vol. 3, no. 3, May 1986, pp. 27-39.

[Balzer84] Balzer, B., "Evolution as a new basis for reusability", *Proceedings, Compsac84 Conference*, Silver Spring, Maryland: IEEE Computer Society Press, 1984, p. 471.

The evolutionary software development paradigm as the basis of reusability is discussed. The differences between the conventional approach and the automation-based evolutionary approach are analyzed. The effects on the alternative paradigm on software reusability are summarized. Rather than reuse of previous implementations, the reuse of the development process steps can be provided.

[Barbuceanu86] Barbuceanu, M., "Knowledge based development of evolutionary and reusable software", *Proceedings, Conference on Expert Systems and Their Applications*, Avignon: 1986, p. 181.

[Berzins86] Berzins, V., Gray, M., Naumann, D., "Abstraction-Based Software Development", *Communications of the ACM*, vol. 29, no. 5, May 1986, pp. 402-415.

[Cheatham84] Cheatham, T.E. Jr., "Reusability Through Program Transformations", *IEEE Transactions on Software Engineering*, vol. SE-10, no. 5, May 1984, pp. 589-594.

A methodology supporting an environment that facilitates the reuse of abstract programs is introduced. Abstract programs are written in a domain-dependent language extended from a base language. The abstract programs are then transformed into concrete software. A number of experiments have been done in the rapid prototyping of abstract programs into their concrete counterparts, as well as in custom tailoring which produces a family of concrete programs from the same abstract counterparts. The environment supporting the approach includes a software database for abstract and concrete programs, a user interface and a tool set to manipulate artifacts in the database.

[Dennis86] Dennis, R. St., Stachour, P., Frankowski, E., and Onuegbe, E., "Measurable characteristics of reusable Ada (R) software", ACM SIGAda *Ada Letters*, vol. VI, no. 2, March, April 1986, pp. 41-50.

The classes of *metacharacteristics* and *characteristics* of reusable software components are discussed. The Ada language is analyzed as far as both of the classes are concerned, and guidelines for reusable Ada software are given.

[Doberkat83] Doberkat, E., Dubinsky, E., Schwartz, J.T., "Reusability of design for complex programs: An experiment with the SETL optimizer", *Proceedings, ITT Workshop on Reusability in Programming*, Stratford, Connecticut: ITT Programming, 1983, p. 106.

[Feather83] Feather, M.S., "Reuse in the context of a transformation based methodology," *Proceedings, ITT Workshop on Reusability in Programming*, Stratford, Connecticut: ITT Programming, 1983, p. 50.

[Freeman86] Freeman, P., "A conceptual analysis of the Draco approach to constructing software systems", University of California, Irvine, Department of Information and Computer Science, RTP042, 1986, 38 p., (To appear in *IEEE Transactions on Software Engineering*).

[Goguen84] Goguen, J.A., "Parameterized Programming," *IEEE Transactions on Software Engineering*, vol. SE-10, no. 5, May 1984, pp. 528-543.

The article suggests that parameterized programming is a powerful technique for reuse. The approach is based on flexible descriptions of general module interfaces that can be instantiated by one or more parameters. Inheritance of these interface specifications is suggested. The new concepts in parameterized programming, *theories*, *views* and *module expressions* are illustrated with examples in the OBJ programming language.

[Goguen86] Goguen, J.A., "Reusing and Interconnecting Software Components," *IEEE Computer*, vol. 19, no. 2, Feb. 1986, pp. 16-28.

[Goodell83] Goodell, M., "Quantitative study of functional commonality in a sample of commercial business applications", *Proceedings, ITT Workshop on Reusability in Programming*, Stratford, Connecticut: ITT Programming, 1983, p. 279.

[Green83] Green, C., Luckham, D., Balzer, R., Cheatham, T., and Rich, C., (eds.), *Report on a knowledge-based software assistant*, Kestrel Institute, Report No. KES.U.83.2, Palo Alto, California, 1983, p. 71.

[Kartashev86] Kartashev, S.P., Kartashev, S.I., "Adaptable Software for Dynamic Architectures," *IEEE Computer*, vol. 19, no. 2, Feb. 1986, pp. 61-77.

[Litvintchouk84] Litvintchouk, S.D., Matsumoto, A.S., "Design of Ada Systems Yielding Reusable Components: An Approach Using Structured Algebraic Specification," *IEEE Transactions on Software Engineering*, vol. SE-10, no. 5, May 1984, pp. 544-551.

A methodology based on formal algebra is suggested to achieve the design and management of reusable components in Ada systems. A specification language, also based on formal algebra, has been developed to express the requirements for the use of Ada packages that are not met by the Ada language itself, but which can then be implemented in Ada. The requirements concern especially intercomponent specifications that allow the use of Ada components in several different applications.

[Lubars86] Lubars, M.D., "Code reusability in the large versus code reusability in the small," ACM SIGSOFT *Software Engineering Notes*, vol 11., no. 1, March, Apr. 1986, pp. 21-28.

Code reusability in the small is defined to mean the reuse of a fragment of a program directly without modifications in another program within the same company or project, on the contrary to *code reusability in the large* that promotes "code sharing" among large numbers of products and programmers. The problems in reusing code in the small and in the large are discussed, as far as the selection and use of the code is concerned. The techniques enhancing code reuse, code templates, schemas and abstractions, are then discussed.

[Neighbors80] Neighbors, J.M., "Software construction using components," Ph.D. thesis, University of California, Irvine, Department of Information and Computer Science, 1980, p. 154.

[Neighbors84] Neighbors, J.M., "The Draco approach to constructing software from reusable components," *IEEE Transactions on Software Engineering*, vol. SE-10, no. 5, May 1984, pp. 564-574.

The paper introduces the Draco approach in reusing software analysis and design results when constructing similar software systems from components. Components in Draco are organized into problem areas or domains, described in a domain language, optimized by source-to-source translations and refined to components in other domains. The reuse of domain analysis is done always when a system is specified in the domain language developed for the specific domain, and the reuse of domain design is done when the domain-dependent components are used to refine the specification into another domain. A domain description includes definitions for *a parser, a prettyprinter, transformations, components* and *procedures*. An example of a domain organization is given and the features of the Draco approach are discussed.

[Prieto-Diaz85] Prieto-Diaz, R., "A software classification scheme," Ph.D. thesis, University of California, Irvine, Department of Information and Computer Science, 1985, p. 194.

A general software classification scheme that organizes reusability related attributes and functions from different domains is proposed as a partial solution to the problems of reuse for code fragments. Evaluation of common characteristics can be used as the basis of reuse of similar, potentially reusable components. A library system with an evaluation mechanism is presented to integrate the proposed classification scheme.

[Prieto-Diaz86] Prieto-Diaz, R., Freeman, P., "A software classification scheme for reusability," *IEEE Software*, vol. 4, no. 1, Jan. 1987, pp. 6-16.

[Polster86] Polster, F.J., "Reuse of Software Through Generation of Partial Systems," *IEEE Transactions on Software Engineering*, vol. SE-12, no. 3, March 1986, pp. 402-416.

The problem of generating partial systems from a subset of general capabilities is addressed. A heuristic to select the existing code fragments needed as the building blocks for the other programs is presented. The notion of "*B*-program" that includes the code fragment and the information about the partial system it is relevant for, is introduced. A formal model for the construction of partial systems from *B*-programs has been developed and is discussed.

[Ramamoorthy86] Ramamoorthy, C.V., Garg, V., and Prakash, A., "Programming in the Large," *IEEE Transactions on Software Engineering*, vol. SE-12, no. 7, July 1986, pp. 769-783.

The *phase-independent* and *phase-dependent* techniques for programming in the large are studied. One of the phase-independent techniques is software reusability. Although the benefits of reusability have been shown by several studies, some of the reusability aspects, *managerial issues, reusability level, reusability metrics* and *software library assistants* are seen to lack sufficient attention in current literature. These aspects are investigated as far as programming in the large of complex software systems is concerned.

[Rich83] Rich, C., Waters, R.C., "Formalizing reusable software components," *Proceedings, ITT Workshop on Reusability in Programming*, Stratford, Connecticut: ITT Programming, 1983, p. 155.

[Roussapoulos84] Roussapoulos, N., Yeh, R.T., "An Adaptable Methodology for Database Design," *IEEE Computer*, vol. 17, no. 5, May 1984, pp. 64-80.

A comprehensive step-by-step methodology for adaptable database design is presented. The methodology is adaptable, because each phase can be facilitated by a number of models and representation primitives. The approach is "outside-in", as the very first phase proposed is *environment* and *requirements analysis*, followed by *systems analysis* and *specification, conceptual data modeling* and *derivation of a logical access path schema* that is the level where existing DBMSs can be taken in use.

[Schwederski86] Schwederski, T., Siegel, H.J., "Adaptable Software for Supercomputers," *IEEE Computer*, vol. 19, no. 2, Feb. 1986, pp. 40-48.

[Soloway84] Soloway, E., Ehrlich, K., "Empirical Studies of Programming Knowledge," *IEEE Transactions on Software Engineering*, vol. SE-10, no. 5, May 1984, pp. 595-609.

The knowledge types used by experienced programmers are studied: *programming plans* are generic program fragments of stereotyping action sequences, *rules of programming discourse* govern the composition of plans into programs. The results are based on empirical studies of a "fill-in-the-blank" problem and a verbatim recall task that are described in detail.

[Terwilliger86] Terwilliger, R.B., Campbell, R.H., "PLEASE: Executable Specifications for Incremental Software Development," Department of Computer Science, University of Illinois, Urbana-Champaign, Report No. UIUCDCS-R-86-1295, 1986, p. 24.